

# Package ‘cmna’

July 7, 2020

**Type** Package

**Title** Computational Methods for Numerical Analysis

**Version** 1.0.3

**Date** 2020-07-05

**Encoding** UTF-8

**URL** <https://jameshoward.us/cmna>

**BugReports** <https://github.com/k3jph/cmna-pkg/issues>

**Description** Provides the source and examples for James P. Howard, II, “Computational Methods for Numerical Analysis with R,” <<https://jameshoward.us/cmna>>, a book on numerical methods in R.

**License** BSD\_2\_clause + file LICENSE

**LazyLoad** no

**Suggests** testthat, devtools, markdown, roxygen2

**Depends** R (>= 2.10)

**RoxygenNote** 7.1.0

**NeedsCompilation** no

**Author** James Howard [aut, cre] (<<https://orcid.org/0000-0003-4530-1547>>)

**Maintainer** James Howard <jh@jameshoward.us>

**Repository** CRAN

**Date/Publication** 2020-07-07 14:50:09 UTC

## R topics documented:

cmna-package . . . . .	3
adaptint . . . . .	3
bezier . . . . .	4
bilinear . . . . .	5
bisection . . . . .	6

bvp	7
choleskymatrix	8
cubicspline	8
detmatrix	9
division	10
fibonacci	11
findiff	12
gaussint	13
gds	14
giniquintile	15
goldsect	16
gradient	17
heat	18
hillclimbing	19
himmelblau	20
horner	20
invmatrix	21
isPrime	22
iterativematrix	23
ivp	24
ivpsys	25
linterp	25
lumatrix	26
mcint	27
midpt	28
newton	29
nn	30
nthroot	31
polyinterp	32
pwiselinterp	33
quadratic	34
refmatrix	35
resizeImage	36
revolution-solid	36
romberg	37
rowops	38
sa	39
secant	40
simp	41
simp38	42
summation	43
trap	44
tridiagmatrix	45
vecnorm	45
wave	46
wilkinson	47

---

cmna-package

*Computational Methods for Numerical Analysis*

---

## Description

Provides the source and examples for *Computational Methods for Numerical Analysis with R*.

## Details

This package provides a suite of simple implementations of standard methods from numerical analysis. The collection is designed to accompany *Computational Methods for Numerical Analysis with R* by James P. Howard, II. Together, these functions provide methods to support linear algebra, interpolation, integration, root finding, optimization, and differential equations.

## Author(s)

James P. Howard, II <jh@jameshoward.us>

## See Also

Useful links:

- <https://jameshoward.us/cmna>
- Report bugs at <https://github.com/k3jph/cmna-pkg/issues>

---

adaptint

*Adaptive Integration*

---

## Description

Adaptive integration

## Usage

```
adaptint(f, a, b, n = 10, tol = 1e-06)
```

## Arguments

f	function to integrate
a	the a-bound of integration
b	the b-bound of integration
n	the maximum recursive depth
tol	the maximum error tolerance

**Details**

The `adaptint` function uses Romberg's rule to calculate the integral of the function `f` over the interval from `a` to `b`. The parameter `n` sets the number of intervals to use when evaluating. Additional options are passed to the function `f` when evaluating.

**Value**

the value of the integral

**See Also**

Other integration: [gaussint\(\)](#), [giniquintile\(\)](#), [mcint\(\)](#), [midpt\(\)](#), [revolution-solid](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

Other newton-cotes: [giniquintile\(\)](#), [midpt\(\)](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

**Examples**

```
f <- function(x) { sin(x)^2 + log(x) }
adaptint(f, 1, 10, n = 4)
adaptint(f, 1, 10, n = 5)
adaptint(f, 1, 10, n = 10)
```

---

bezier

*Bezier curves*


---

**Description**

Find the quadratic and cubic Bezier curve for the given points

**Usage**

```
qbezier(x, y, t)
```

```
cbezier(x, y, t)
```

**Arguments**

<code>x</code>	a vector of x values
<code>y</code>	a vector of y values
<code>t</code>	a vector of t values for which the curve will be computed

**Details**

`qbezier` finds the quadratic Bezier curve for the given three points and `cbezier` finds the cubic Bezier curve for the given four points. The curve will be computed at all values in the vector `t` and a list of `x` and `y` values returned.

**Value**

a list composed of an x-vector and a y-vector

**See Also**

Other interp: [bilinear\(\)](#), [cubicspline\(\)](#), [linterp\(\)](#), [nn\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#)

**Examples**

```
x <- c(1, 2, 3)
y <- c(2, 3, 5)
f <- qbezier(x, y, seq(0, 1, 1/100))
```

```
x <- c(-1, 1, 0, -2)
y <- c(-2, 2, -1, -1)
f <- cbezier(x, y, seq(0, 1, 1/100))
```

---

bilinear

*Bilinear interpolation*

---

**Description**

Finds a bilinear interpolation bounded by four points

**Usage**

```
bilinear(x, y, z, newx, newy)
```

**Arguments**

x	vector of two x values representing x_1 and x_2
y	vector of two y values representing y_1 and y_2
z	2x2 matrix if z values
newx	vector of new x values to interpolate
newy	vector of new y values to interpolate

**Details**

bilinear finds a bilinear interpolation bounded by four corners

**Value**

a vector of interpolated z values at (x, y)

**See Also**

Other interp: [bezier](#), [cubicspline\(\)](#), [linterp\(\)](#), [nn\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#)

Other algebra: [cubicspline\(\)](#), [division](#), [fibonacci\(\)](#), [horner\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

**Examples**

```
x <- c(2, 4)
y <- c(4, 7)
z <- matrix(c(81, 84, 85, 89), nrow = 2)
newx <- c(2.5, 3, 3.5)
newy <- c(5, 5.5, 6)
bilinear(x, y, z, newx, newy)
```

---

bisection

*The Bisection Method*


---

**Description**

Use the bisection method to find real roots

**Usage**

```
bisection(f, a, b, tol = 0.001, m = 100)
```

**Arguments**

f	function to locate a root for
a	the a bound of the search region
b	the b bound of the search region
tol	the error tolerance
m	the maximum number of iterations

**Details**

The bisection method functions by repeatedly halving the interval between a and b and will return when the interval between them is less than tol, the error tolerance. However, this implementation also stops if after m iterations.

**Value**

the real root found

**See Also**

Other optimz: [goldsect](#), [gradient](#), [hillclimbing\(\)](#), [newton\(\)](#), [sa\(\)](#), [secant\(\)](#)

**Examples**

```
f <- function(x) { x^3 - 2 * x^2 - 159 * x - 540 }
bisection(f, 0, 10)
```

---

**bvp***Boundary value problems*

---

**Description**

solve boundary value problems for ordinary differential equations

**Usage**

```
bvpexample(x)
bvpexample10(x)
```

**Arguments**

x                    proposed initial x-value

**Details**

The euler method implements the Euler method for solving differential equations. The codemidp-  
tvp method solves initial value problems using the second-order Runge-Kutta method. The rungekutta4  
method is the fourth-order Runge-Kutta method.

**Value**

a data frame of x and y values

**Examples**

```
bvpexample(-2)
bvpexample(-1)
bvpexample(0)
bvpexample(1)
bvpexample(2)
## (bvp.b <- bisection(bvpexample, 0, 1))
## (bvp.s <- secant(bvpexample, 0))
```

choleskymatrix      *Cholesky Decomposition*

---

**Description**

Decompose a matrix into the Cholesky

**Usage**

```
choleskymatrix(m)
```

**Arguments**

m                    a matrix

**Details**

choleskymatrix decomposes the matrix m into the LU decomposition, such that  $m == L$

**Value**

the matrix L

**See Also**

Other linear: [detmatrix\(\)](#), [gdls\(\)](#), [invmatrix\(\)](#), [iterativematrix](#), [lumatrix\(\)](#), [refmatrix\(\)](#), [rowops](#), [tridiagmatrix\(\)](#), [vecnorm\(\)](#)

**Examples**

```
(A <- matrix(c(5, 1, 2, 1, 9, 3, 2, 3, 7), 3))
(L <- choleskymatrix(A))
t(L) %**% L
```

---

cubicspline      *Natural cubic spline interpolation*

---

**Description**

Finds a piecewise linear function that interpolates the data points

**Usage**

```
cubicspline(x, y)
```



**Arguments**

x                    a vector of x values  
y                    a vector of y values

**Details**

cubicspline finds a piecewise cubic spline function that interpolates the data points. For each x-y ordered pair. The function will return a list of four vectors representing the coefficients.

**Value**

a list of coefficient vectors

**See Also**

Other interp: [bezier](#), [bilinear\(\)](#), [linterp\(\)](#), [nn\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#)

Other algebra: [bilinear\(\)](#), [division](#), [fibonacci\(\)](#), [horner\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

**Examples**

```
x <- c(1, 2, 3)
y <- c(2, 3, 5)
f <- cubicspline(x, y)
```

```
x <- c(-1, 1, 0, -2)
y <- c(-2, 2, -1, -1)
f <- cubicspline(x, y)
```

---

detmatrix

*Calculate the determinant of the matrix*

---

**Description**

Calculate the determinant of the matrix

**Usage**

```
detmatrix(m)
```

**Arguments**

m                    a matrix

**Details**

detmatrix calculates the determinant of the matrix given.

**Value**

the determinant

**See Also**

Other linear: [choleskymatrix\(\)](#), [gdls\(\)](#), [invmatrix\(\)](#), [iterativematrix](#), [lumatrix\(\)](#), [refmatrix\(\)](#), [rowops](#), [tridiagmatrix\(\)](#), [vecnorm\(\)](#)

**Examples**

```
A <- matrix(c(1, 2, -7, -1, -1, 1, 2, 1, 5), 3)
detmatrix(A)
```

---

division

*Algorithms for divisions*

---

**Description**

Algorithms for division that provide a quotient and remainder.

**Usage**

```
naivediv(m, n)
```

```
longdiv(m, n)
```

**Arguments**

m                   the dividend

n                    the divisor

**Details**

The `naivediv` divides `m` by `n` by using repeated division. The `longdiv` function uses the long division algorithm in binary.

**Value**

the quotient and remainder as a list

**See Also**

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [fibonacci\(\)](#), [horner\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

**Examples**

```
a <- floor(runif(1, 1, 1000))
b <- floor(runif(1, 1, 100))
naivediv(a, b)
longdiv(a, b)
```

---

fibonacci

*Fibonacci numbers*

---

**Description**

Return the n-th Fibonacci number

**Usage**

```
fibonacci(n)
```

**Arguments**

n                    n

**Details**

This function recursively implements the famous Fibonacci sequence. The function returns the nth member of the sequence.

**Value**

the sequence element

**See Also**

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [division](#), [horner\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

**Examples**

```
fibonacci(10)
```

---

`findiff`*Finite Differences*

---

**Description**

Finite differences formulas

**Usage**

```
findiff(f, x, h = x * sqrt(.Machine$double.eps))
```

```
syndiff(f, x, h = x * .Machine$double.eps^(1/3))
```

```
findiff2(f, x, h)
```

```
rdiff(f, x, n = 10, h = 1e-04)
```

**Arguments**

<code>f</code>	function to differentiate
<code>x</code>	the x-value to differentiate at
<code>h</code>	the step-size for evaluation
<code>n</code>	the maximum number of convergence steps in <code>rdiff</code>

**Details**

The `findiff` formula uses the finite differences formula to find the derivative of `f` at `x`. The value of `h` is the step size of the evaluation. The function `findiff2` provides the second derivative.

**Value**

the value of the derivative

**Examples**

```
findiff(sin, pi, 1e-3)
syndiff(sin, pi, 1e-3)
```

---

`gaussint`*Gaussian integration method driver*

---

**Description**

Use the Gaussian method to evaluate integrals

**Usage**

```
gaussint(f, x, w)
```

```
gauss.legendre(f, m = 5)
```

```
gauss.laguerre(f, m = 5)
```

```
gauss.hermite(f, m = 5)
```

**Arguments**

<code>f</code>	function to integrate
<code>x</code>	list of evaluation points
<code>w</code>	list of weights
<code>m</code>	number of evaluation points

**Details**

The `gaussint` function uses the Gaussian integration to evaluate an integral. The function itself is a driver and expects the integration points and associated weights as options.

**Value**

the value of the integral

**See Also**

Other integration: [adaptint\(\)](#), [giniquintile\(\)](#), [mcint\(\)](#), [midpt\(\)](#), [revolution-solid](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

**Examples**

```
w = c(1, 1)
x = c(-1 / sqrt(3), 1 / sqrt(3))
f <- function(x) { x^3 + x + 1 }
gaussint(f, x, w)
```

---

`gdls`*Least squares with gradient descent*

---

**Description**

Solve least squares with gradient descent

**Usage**

```
gdls(A, b, alpha = 0.05, tol = 1e-06, m = 1e+05)
```

**Arguments**

<code>A</code>	a square matrix representing the coefficients of a linear system
<code>b</code>	a vector representing the right-hand side of the linear system
<code>alpha</code>	the learning rate
<code>tol</code>	the expected error tolerance
<code>m</code>	the maximum number of iterations

**Details**

`gdls` solves a linear system using gradient descent.

**Value**

the modified matrix

**See Also**

Other linear: [choleskymatrix\(\)](#), [detmatrix\(\)](#), [invmatrix\(\)](#), [iterativematrix](#), [lumatrix\(\)](#), [refmatrix\(\)](#), [rowops](#), [tridiagmatrix\(\)](#), [vecnorm\(\)](#)

**Examples**

```
head(b <- iris$Sepal.Length)
head(A <- matrix(cbind(1, iris$Sepal.Width, iris$Petal.Length, iris$Petal.Width), ncol = 4))
gdls(A, b, alpha = 0.05, m = 10000)
```

---

giniquintile	<i>Gini coefficients</i>
--------------	--------------------------

---

**Description**

Calculate the Gini coefficient from quintile data

**Usage**

```
giniquintile(L)
```

**Arguments**

L                    vector of percentages at 20th, 40th, 60th, and 80th percentiles

**Details**

Calculate the Gini coefficient given the quintile data.

**Value**

the estimated Gini coefficient

**References**

Leon Gerber, "A Quintile Rule for the Gini Coefficient", *Mathematics Magazine*, 80:2, April 2007.

**See Also**

Other integration: [adaptint\(\)](#), [gaussint\(\)](#), [mcint\(\)](#), [midpt\(\)](#), [revolution-solid](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

Other newton-cotes: [adaptint\(\)](#), [midpt\(\)](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

**Examples**

```
L <- c(4.3, 9.8, 15.4, 22.7)
giniquintile(L)
```

---

`goldsect`*Golden Section Search*

---

**Description**

Use golden section search to find local extrema

**Usage**

```
goldsectmin(f, a, b, tol = 0.001, m = 100)
```

```
goldsectmax(f, a, b, tol = 0.001, m = 100)
```

**Arguments**

<code>f</code>	function to integrate
<code>a</code>	the a bound of the search region
<code>b</code>	the b bound of the search region
<code>tol</code>	the error tolerance
<code>m</code>	the maximum number of iterations

**Details**

The golden section search method functions by repeatedly dividing the interval between a and b and will return when the interval between them is less than `tol`, the error tolerance. However, this implementation also stop if after `m` iterations.

**Value**

the x value of the minimum found

**See Also**

Other optimz: [bisection\(\)](#), [gradient](#), [hillclimbing\(\)](#), [newton\(\)](#), [sa\(\)](#), [secant\(\)](#)

**Examples**

```
f <- function(x) { x^2 - 3 * x + 3 }  
goldsectmin(f, 0, 5)
```



---

gradient	<i>Gradient descent</i>
----------	-------------------------

---

### Description

Use gradient descent to find local minima

### Usage

```
graddsc(fp, x, h = 0.001, tol = 1e-04, m = 1000)
```

```
gradasc(fp, x, h = 0.001, tol = 1e-04, m = 1000)
```

```
gd(fp, x, h = 100, tol = 1e-04, m = 1000)
```

### Arguments

fp	function representing the derivative of f
x	an initial estimate of the minima
h	the step size
tol	the error tolerance
m	the maximum number of iterations

### Details

Gradient descent can be used to find local minima of functions. It will return an approximation based on the step size  $h$  and  $fp$ . The  $tol$  is the error tolerance,  $x$  is the initial guess at the minimum. This implementation also stops after  $m$  iterations.

### Value

the  $x$  value of the minimum found

### See Also

Other optimz: [bisection\(\)](#), [goldsect](#), [hillclimbing\(\)](#), [newton\(\)](#), [sa\(\)](#), [secant\(\)](#)

### Examples

```
fp <- function(x) { x^3 + 3 * x^2 - 1 }  
graddsc(fp, 0)
```

```
f <- function(x) { (x[1] - 1)^2 + (x[2] - 1)^2 }  
fp <-function(x) {  
  x1 <- 2 * x[1] - 2  
  x2 <- 8 * x[2] - 8
```

```
    return(c(x1, x2))
  }
gd(fp, c(0, 0), 0.05)
```

---

heat

*Heat Equation via Forward-Time Central-Space*

---

### Description

solve heat equation via forward-time central-space method

### Usage

```
heat(u, alpha, xdelta, tdelta, n)
```

### Arguments

u	the initial values of u
alpha	the thermal diffusivity coefficient
xdelta	the change in x at each step in u
tdelta	the time step
n	the number of steps to take

### Details

The heat solves the heat equation using the forward-time central-space method in one-dimension.

### Value

a matrix of u values at each time step

### Examples

```
alpha <- 1
x0 <- 0
xdelta <- .05
x <- seq(x0, 1, xdelta)
u <- sin(x^4 * pi)
tdelta <- .001
n <- 25
z <- heat(u, alpha, xdelta, tdelta, n)
```

---

hillclimbing	<i>Hill climbing</i>
--------------	----------------------

---

**Description**

Use hill climbing to find the global minimum

**Usage**

```
hillclimbing(f, x, h = 1, m = 1000)
```

**Arguments**

f	function representing the derivative of f
x	an initial estimate of the minimum
h	the step size
m	the maximum number of iterations

**Details**

Hill climbing

**Value**

the x value of the minimum found

**See Also**

Other optimz: [bisection\(\)](#), [goldsect](#), [gradient](#), [newton\(\)](#), [sa\(\)](#), [secant\(\)](#)

**Examples**

```
f <- function(x) {
  (x[1]^2 + x[2] - 11)^2 + (x[1] + x[2]^2 - 7)^2
}
hillclimbing(f, c(0,0))
hillclimbing(f, c(-1,-1))
hillclimbing(f, c(10,10))
```

---

`himmelblau`*Himmelblau Function*

---

**Description**

Generate the Himmelblau function

**Usage**`himmelblau(x)`**Arguments**

`x` a vector of x-values

**Details**

Generate the Himmelblau function

**Value**

the value of the function at `x`.

---

`horner`*Horner's rule*

---

**Description**

Use Horner's rule to evaluate a polynomial

**Usage**`horner(x, coefs)``rhorer(x, coefs)``naivepoly(x, coefs)``betterpoly(x, coefs)`**Arguments**

`x` a vector of x values to evaluate the polynomial

`coefs` vector of coefficients of x

**Details**

This function implements Horner's rule for fast polynomial evaluation. The implementation expects  $x$  to be a vector of  $x$  values at which to evaluate the polynomial. The parameter `coefs` is a vector of coefficients of  $x$ . The vector order is such that the first element is the constant term, the second element is the coefficient of  $x$ , the so forth to the highest degreed term. Terms with a 0 coefficient should have a 0 element in the vector.

The function `rhorners` implements the the Horner algorithm recursively.

The function `naivepoly` implements a polynomial evaluator using the straightforward algebraic approach.

The function `betterpoly` implements a polynomial evaluator using the straightforward algebraic approach with cached  $x$  terms.

**Value**

the value of the function at  $x$

**See Also**

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [division](#), [fibonacci\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

**Examples**

```
b <- c(2, 10, 11)
x <- 5
horner(x, b)
b <- c(-1, 0, 1)
x <- c(1, 2, 3, 4)
horner(x, b)
rhorners(x, b)
```

---

 invmatrix

*Invert a matrix*


---

**Description**

Invert the matrix using Gaussian elimination

**Usage**

```
invmatrix(m)
```

**Arguments**

`m` a matrix

**Details**

`invmatrix` invertse the given matrix using Gaussian elimination and returns the result.

**Value**

the inverted matrix

**See Also**

Other linear: [choleskymatrix\(\)](#), [detmatrix\(\)](#), [gdls\(\)](#), [iterativematrix](#), [lumatrix\(\)](#), [refmatrix\(\)](#), [rowops](#), [tridiagmatrix\(\)](#), [vecnorm\(\)](#)

**Examples**

```
A <- matrix(c(1, 2, -7, -1, -1, 1, 2, 1, 5), 3)
refmatrix(A)
```

---

isPrime

*Test for Primality*

---

**Description**

Test the number given for primality.

**Usage**

```
isPrime(n)
```

**Arguments**

n                    n

**Details**

This function tests n if it is prime through repeated division attempts. If a match is found, by finding a remainder of 0, FALSE is returned.

**Value**

boolean TRUE if n is prime, FALSE if not

**See Also**

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [division](#), [fibonacci\(\)](#), [horner\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

## Examples

```
isPrime(37)
isPrime(89)
isPrime(100)
```

---

iterativematrix	<i>Solve a matrix using iterative methods</i>
-----------------	---

---

## Description

Solve a matrix using iterative methods.

## Usage

```
jacobi(A, b, tol = 1e-06, maxiter = 100)
gaussseidel(A, b, tol = 1e-06, maxiter = 100)
cgmmatrix(A, b, tol = 1e-06, maxiter = 100)
```

## Arguments

A	a square matrix representing the coefficients of a linear system
b	a vector representing the right-hand side of the linear system
tol	is a number representing the error tolerance
maxiter	is the maximum number of iterations

## Details

`jacobi` finds the solution using Jacobi iteration. Jacobi iteration depends on the matrix being diagonally-dominant. The tolerance is specified the norm of the solution vector.

`gaussseidel` finds the solution using Gauss-Seidel iteration. Gauss-Seidel iteration depends on the matrix being either diagonally-dominant or symmetric and positive definite.

`cgmmatrix` finds the solution using the conjugate gradient method. The conjugate gradient method depends on the matrix being symmetric and positive definite.

## Value

the solution vector

## See Also

Other linear: [choleskymatrix\(\)](#), [detmatrix\(\)](#), [gdls\(\)](#), [invmatrix\(\)](#), [lumatrix\(\)](#), [refmatrix\(\)](#), [rowops](#), [tridiagmatrix\(\)](#), [vecnorm\(\)](#)

**Examples**

```
A <- matrix(c(5, 2, 1, 2, 7, 3, 3, 4, 8), 3)
b <- c(40, 39, 55)
jacobi(A, b)
```

---

ivp

*Initial value problems*

---

**Description**

solve initial value problems for ordinary differential equations

**Usage**

```
euler(f, x0, y0, h, n)
midptivp(f, x0, y0, h, n)
rungekutta4(f, x0, y0, h, n)
adamsbashforth(f, x0, y0, h, n)
```

**Arguments**

f	function to integrate
x0	the initial value of x
y0	the initial value of y
h	selected step size
n	the number of steps

**Details**

The `euler` method implements the Euler method for solving differential equations. The `codemidptivp` method solves initial value problems using the second-order Runge-Kutta method. The `rungekutta4` method is the fourth-order Runge-Kutta method.

**Value**

a data frame of x and y values

**Examples**

```
f <- function(x, y) { y / (2 * x + 1) }
ivp.euler <- euler(f, 0, 1, 1/100, 100)
ivp.midpt <- midptivp(f, 0, 1, 1/100, 100)
ivp.rk4 <- rungekutta4(f, 0, 1, 1/100, 100)
```



---

`ivpsys`*Initial value problems for systems of ordinary differential equations*

---

**Description**

solve initial value problems for systems ordinary differential equations

**Usage**

```
eulersys(f, x0, y0, h, n)
```

**Arguments**

<code>f</code>	function to integrate
<code>x0</code>	the initial value of x
<code>y0</code>	the vector initial values of y
<code>h</code>	selected step size
<code>n</code>	the number of steps

**Details**

The euler method implements the Euler method for solving differential equations. The `codemidp` method solves initial value problems using the second-order Runge-Kutta method. The `rungekutta4` method is the fourth-order Runge-Kutta method.

**Value**

a data frame of x and y values

**Examples**

```
f <- function(x, y) { y / (2 * x + 1) }  
ivp.euler <- euler(f, 0, 1, 1/100, 100)
```

---

`linterp`*Linear interpolation*

---

**Description**

Finds a linear function between two points

**Usage**

```
linterp(x1, y1, x2, y2)
```

**Arguments**

x1	x value of the first point
y1	y value of the first point
x2	x value of the second point
y2	y value of the second point

**Details**

linterp finds a linear function between two points.

**Value**

a linear equation's coefficients

**See Also**

Other interp: [bezier](#), [bilinear\(\)](#), [cubicspline\(\)](#), [nn\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#)

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [division](#), [fibonacci\(\)](#), [horner\(\)](#), [isPrime\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

**Examples**

```
f <- linterp(3, 2, 7, -2)
```

---

 lumatrix

*LU Decomposition*


---

**Description**

Decompose a matrix into lower- and upper-triangular matrices

**Usage**

```
lumatrix(m)
```

**Arguments**

m	a matrix
---	----------

**Details**

lumatrix decomposes the matrix m into the LU decomposition, such that  $m == L$

**Value**

list with matrices L and U representing the LU decomposition

**See Also**

Other linear: [choleskymatrix\(\)](#), [detmatrix\(\)](#), [gdls\(\)](#), [invmatrix\(\)](#), [iterativematrix](#), [refmatrix\(\)](#), [rowops](#), [tridiagmatrix\(\)](#), [vecnorm\(\)](#)

**Examples**

```
A <- matrix(c(1, 2, -7, -1, -1, 1, 2, 1, 5), 3)
lumatrix(A)
```

---

 mcint

*Monte Carlo Integration*


---

**Description**

Simple Monte Carlo Integraton

**Usage**

```
mcint(f, a, b, m = 1000)

mcint2(f, xdom, ydom, m = 1000)
```

**Arguments**

f	function to integrate
a	the lower-bound of integration
b	the upper-bound of integration
m	the number of subintervals to calculate
xdom	the domain on x of integration in two dimensions
ydom	the domain on y of integration in two dimensions

**Details**

The mcint function uses a simple Monte Carlo algorithm to estimate the value of an integral. The parameter n sets the total number of evaluation points. The parameter max.y is the maximum expected value of the range of function f. The mcint2 provides Monte Carlo integration in two dimensions.

**Value**

the value of the integral

**See Also**

Other integration: [adaptint\(\)](#), [gaussint\(\)](#), [giniquintile\(\)](#), [midpt\(\)](#), [revolution-solid](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

**Examples**

```
f <- function(x) { sin(x)^2 + log(x) }
mcint(f, 0, 1)
mcint(f, 0, 1, m = 10e6)
```

---

midpt	<i>rectangle method</i>
-------	-------------------------

---

**Description**

Use the rectangle method to integrate a function

**Usage**

```
midpt(f, a, b, m = 100)
```

**Arguments**

f	function to integrate
a	the a-bound of integration
b	the b-bound of integration
m	the number of subintervals to calculate

**Details**

The midpt function uses the rectangle method to calculate the integral of the function f over the interval from a to b. The parameter m sets the number of intervals to use when evaluating the rectangles. Additional options are passed to the function f when evaluating.

**Value**

the value of the integral

**See Also**

Other integration: [adaptint\(\)](#), [gaussint\(\)](#), [giniquintile\(\)](#), [mcint\(\)](#), [revolution-solid](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

Other newton-cotes: [adaptint\(\)](#), [giniquintile\(\)](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

**Examples**

```
f <- function(x) { sin(x)^2 + cos(x)^2 }
midpt(f, -pi, pi, m = 10)
midpt(f, -pi, pi, m = 100)
midpt(f, -pi, pi, m = 1000)
```

---

newton	<i>Newton's method</i>
--------	------------------------

---

### Description

Use Newton's method to find real roots

### Usage

```
newton(f, fp, x, tol = 0.001, m = 100)
```

### Arguments

f	function to integrate
fp	function representing the derivative of f
x	an initial estimate of the root
tol	the error tolerance
m	the maximum number of iterations

### Details

Newton's method finds real roots of a function, but requires knowing the function derivative. It will return when the interval between them is less than `tol`, the error tolerance. However, this implementation also stops after `m` iterations.

### Value

the real root found

### See Also

Other optimz: [bisection\(\)](#), [goldsect](#), [gradient](#), [hillclimbing\(\)](#), [sa\(\)](#), [secant\(\)](#)

### Examples

```
f <- function(x) { x^3 - 2 * x^2 - 159 * x - 540 }
fp <- function(x) { 3 * x^2 - 4 * x - 159 }
newton(f, fp, 1)
```

---

nn	<i>Nearest interpolation</i>
----	------------------------------

---

### Description

Find the nearest neighbor for a set of data points

### Usage

```
nn(p, y, q)
```

### Arguments

p	matrix of variable values, each row is a data point
y	vector of values, each entry corresponds to one row in p
q	vector of variable values, each entry corresponds to one column of p

### Details

nn finds the n-dimensional nearest neighbor for given datapoint

### Value

an interpolated value for *q*

### See Also

Other interp: [bezier](#), [bilinear\(\)](#), [cubicspline\(\)](#), [linterp\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#)

### Examples

```
p <- matrix(floor(runif(100, 0, 9)), 20)
y <- floor(runif(20, 0, 9))
q <- matrix(floor(runif(5, 0, 9)), 1)
nn(p, y, q)
```

---

nthroot	<i>The n-th root formula</i>
---------	------------------------------

---

### Description

Find the n-th root of real numbers

### Usage

```
nthroot(a, n, tol = 1/1000)
```

### Arguments

a	a positive real number
n	n
tol	the permitted error tolerance

### Details

The nthroot function finds the nth root of a via an iterative process.

### Value

the root

### See Also

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [division](#), [fibonacci\(\)](#), [horner\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

### Examples

```
nthroot(100, 2)
nthroot(65536, 4)
nthroot(1000, 3)
```

---

polyinterp	<i>Polynomial interpolation</i>
------------	---------------------------------

---

**Description**

Finds a polynomial function interpolating the given points

**Usage**

```
polyinterp(x, y)
```

**Arguments**

x	a vector of x values
y	a vector of y values

**Details**

polyinterp finds a polynomial that interpolates the given points.

**Value**

a polynomial equation's coefficients

**See Also**

Other interp: [bezier](#), [bilinear\(\)](#), [cubicspline\(\)](#), [linterp\(\)](#), [nn\(\)](#), [pwiselinterp\(\)](#)

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [division](#), [fibonacci\(\)](#), [horner\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [pwiselinterp\(\)](#), [quadratic\(\)](#)

**Examples**

```
x <- c(1, 2, 3)
y <- x^2 + 5 * x - 3
f <- polyinterp(x, y)
```



---

pwiselinterp

*Piecewise linear interpolation*

---

## Description

Finds a piecewise linear function that interpolates the data points

## Usage

```
pwiselinterp(x, y)
```

## Arguments

x	a vector of x values
y	a vector of y values

## Details

pwiselinterp finds a piecewise linear function that interpolates the data points. For each x-y ordered pair, there function finds the unique line interpolating them. The function will return a data.frame with three columns.

The column x is the upper bound of the domain for the given piece. The columns m and b represent the coefficients from the y-intercept form of the linear equation,  $y = mx + b$ .

The matrix will contain length(x) rows with the first row having m and b of NA.

## Value

a matrix with the linear function components

## See Also

Other interp: [bezier](#), [bilinear\(\)](#), [cubicspline\(\)](#), [linterp\(\)](#), [nn\(\)](#), [polyinterp\(\)](#)

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [division](#), [fibonacci\(\)](#), [horner\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [quadratic\(\)](#)

## Examples

```
x <- c(5, 0, 3)
y <- c(4, 0, 3)
f <- pwiselinterp(x, y)
```

---

`quadratic`*The quadratic equation.*

---

**Description**

Find the zeros of a quadratic equation.

**Usage**

```
quadratic(b2, b1, b0)
```

```
quadratic2(b2, b1, b0)
```

**Arguments**

<code>b2</code>	the coefficient of the $x^2$ term
<code>b1</code>	the coefficient of the $x$ term
<code>b0</code>	the constant term

**Details**

`quadratic` and `quadratic2` implement the quadratic equation from standard algebra in two different ways. The `quadratic` function is susceptible to cascading numerical error and the `quadratic2` has reduced potential error.

**Value**

numeric vector of solutions to the equation

**See Also**

Other algebra: [bilinear\(\)](#), [cubicspline\(\)](#), [division](#), [fibonacci\(\)](#), [horner\(\)](#), [isPrime\(\)](#), [linterp\(\)](#), [nthroot\(\)](#), [polyinterp\(\)](#), [pwiselinterp\(\)](#)

**Examples**

```
quadratic(1, 0, -1)
quadratic(4, -4, 1)
quadratic2(1, 0, -1)
quadratic2(4, -4, 1)
```

---

`refmatrix`*Matrix to Row Echelon Form*

---

**Description**

Transform a matrix to row echelon form.

**Usage**`refmatrix(m)``rrefmatrix(m)``solvematrix(A, b)`**Arguments**

`m` a matrix

`A` a square matrix representing the coefficients of a linear system in `solvematrix`

`b` a vector representing the right-hand side of the linear system in `solvematrix`

**Details**

`refmatrix` reduces a matrix to row echelon form. This is not a reduced row echelon form, though that can be easily calculated from the diagonal. This function works on non-square matrices.

`rrefmatrix` returns the reduced row echelon matrix.

`solvematrix` solves a linear system using `rrefmatrix`.

**Value**

the modified matrix

**See Also**

Other linear: [choleskymatrix\(\)](#), [detmatrix\(\)](#), [gdls\(\)](#), [invmatrix\(\)](#), [iterativematrix](#), [lumatrix\(\)](#), [rowops](#), [tridiagmatrix\(\)](#), [vecnorm\(\)](#)

**Examples**

```
A <- matrix(c(1, 2, -7, -1, -1, 1, 2, 1, 5), 3)
refmatrix(A)
```

---

resizeImage	<i>Image resizing</i>
-------------	-----------------------

---

**Description**

Resize images using nearest neighbor and

**Usage**

```
resizeImageNN(imx, width, height)
```

```
resizeImageBL(imx, width, height)
```

**Arguments**

imx	a 3-dimensional array containing image data
width	the new width
height	the new height

**Details**

The *resizeImageNN* function uses the nearest neighbor method to resize the image. Also, *resizeImageBL* uses bilinear interpolation to resize the image.

**Value**

a three-dimensional array containing the resized image.

---

revolution-solid	<i>Volumes of solids of revolution</i>
------------------	--

---

**Description**

Find the volume of a solid of revolution

**Usage**

```
shellmethod(f, a, b)
```

```
discmethod(f, a, b)
```

**Arguments**

f	function of revolution
a	lower-bound of the solid
b	upper-bound of the solid

**Details**

The functions `discmethod` and `shellmethod` implement the algorithms for finding the volume of solids of revolution. The `discmethod` function is suitable for volumes revolved around the x-axis and the `shellmethod` function is suitable for volumes revolved around the y-axis.

**Value**

the volume of the solid

**See Also**

Other integration: [adaptint\(\)](#), [gaussint\(\)](#), [giniquintile\(\)](#), [mcint\(\)](#), [midpt\(\)](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

**Examples**

```
f <- function(x) { x^2 }
shellmethod(f, 1, 2)
discmethod(f, 1, 2)
```

---

romberg

*Romberg Integration*

---

**Description**

Romberg's adaptive integration

**Usage**

```
romberg(f, a, b, m, tab = FALSE)
```

**Arguments**

f	function to integrate
a	the lowerbound of integration
b	the upperbound of integration
m	the maximum number of iterations
tab	if TRUE, return the table of values

**Details**

The `romberg` function uses Romberg's rule to calculate the integral of the function `f` over the interval from `a` to `b`. The parameter `m` sets the number of intervals to use when evaluating. Additional options are passed to the function `f` when evaluating.

**Value**

the value of the integral

**See Also**

Other integration: [adaptint\(\)](#), [gaussint\(\)](#), [giniquintile\(\)](#), [mcint\(\)](#), [midpt\(\)](#), [revolution-solid](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

Other newton-cotes: [adaptint\(\)](#), [giniquintile\(\)](#), [midpt\(\)](#), [simp38\(\)](#), [simp\(\)](#), [trap\(\)](#)

**Examples**

```
f <- function(x) { sin(x)^2 + log(x) }
romberg(f, 1, 10, m = 3)
romberg(f, 1, 10, m = 5)
romberg(f, 1, 10, m = 10)
```

---

rowops

*Elementary row operations*


---

**Description**

These are elementary operations for a matrix. They do not presume a square matrix and will work on any matrix. They use R's internal row addressing to function.

**Usage**

```
swaprows(m, row1, row2)
```

```
replacerow(m, row1, row2, k)
```

```
scalerow(m, row, k)
```

**Arguments**

m	a matrix
row1	a source row
row2	a destination row
k	a scaling factor
row	a row to modify

**Details**

replacerow replaces one row with the sum of itself and the multiple of another row. swaprows swap two rows in the matrix. scalerow scales all entries in a row by a constant.

**Value**

the modified matrix

**See Also**

Other linear: [choleskymatrix\(\)](#), [detmatrix\(\)](#), [gdls\(\)](#), [invmatrix\(\)](#), [iterativematrix](#), [lumatrix\(\)](#), [refmatrix\(\)](#), [tridiagmatrix\(\)](#), [vecnorm\(\)](#)

**Examples**

```
n <- 5
A <- matrix(sample.int(10, n^2, TRUE) - 1, n)
A <- swaprows(A, 2, 4)
A <- replacerow(A, 1, 3, 2)
A <- scalerow(A, 5, 10)
```

---

sa

*Simulated annealing*


---

**Description**

Use simulated annealing to find the global minimum

**Usage**

```
sa(f, x, temp = 10000, rate = 1e-04)
```

```
tspsa(x, temp = 100, rate = 1e-04)
```

**Arguments**

f	function representing f
x	an initial estimate of the minimum
temp	the initial temperature
rate	the cooling rate

**Details**

Simulated annealing finds a global minimum by mimicing the metallurgical process of annealing.

**Value**

the x value of the minimum found

**See Also**

Other optimz: [bisection\(\)](#), [goldsect](#), [gradient](#), [hillclimbing\(\)](#), [newton\(\)](#), [secant\(\)](#)

**Examples**

```
f <- function(x) { x^6 - 4 * x^5 - 7 * x^4 + 22 * x^3 + 24 * x^2 + 2 }
sa(f, 0)
```

```
f <- function(x) { (x[1] - 1)^2 + (x[2] - 1)^2 }
sa(f, c(0, 0), 0.05)
```

---

 secant

*Secant Method*


---

**Description**

The secant method for root finding

**Usage**

```
secant(f, x, tol = 0.001, m = 100)
```

**Arguments**

f	function to integrate
x	an initial estimate of the root
tol	the error tolerance
m	the maximum number of iterations

**Details**

The secant method for root finding extends Newton's method to estimate the derivative. It will return when the interval between them is less than `tol`, the error tolerance. However, this implementation also stop if after `m` iterations.

**Value**

the real root found

**See Also**

Other optimz: [bisection\(\)](#), [goldsect](#), [gradient](#), [hillclimbing\(\)](#), [newton\(\)](#), [sa\(\)](#)

**Examples**

```
f <- function(x) { x^3 - 2 * x^2 - 159 * x - 540 }
secant(f, 1)
```



---

simp	<i>Simpson's rule</i>
------	-----------------------

---

## Description

Use Simpson's rule to integrate a function

## Usage

```
simp(f, a, b, m = 100)
```

## Arguments

f	function to integrate
a	the a-bound of integration
b	the b-bound of integration
m	the number of subintervals to calculate

## Details

The `simp` function uses Simpson's rule to calculate the integral of the function `f` over the interval from `a` to `b`. The parameter `m` sets the number of intervals to use when evaluating. Additional options are passed to the function `f` when evaluating.

## Value

the value of the integral

## See Also

Other integration: [adaptint\(\)](#), [gaussint\(\)](#), [giniquintile\(\)](#), [mcint\(\)](#), [midpt\(\)](#), [revolution-solid](#), [romberg\(\)](#), [simp38\(\)](#), [trap\(\)](#)

Other newton-cotes: [adaptint\(\)](#), [giniquintile\(\)](#), [midpt\(\)](#), [romberg\(\)](#), [simp38\(\)](#), [trap\(\)](#)

## Examples

```
f <- function(x) { sin(x)^2 + cos(x)^2 }  
simp(f, -pi, pi, m = 10)  
simp(f, -pi, pi, m = 100)  
simp(f, -pi, pi, m = 1000)
```

---

`simp38`*Simpson's 3/8 rule*

---

**Description**

Use Simpson's 3/8 rule to integrate a function

**Usage**

```
simp38(f, a, b, m = 100)
```

**Arguments**

<code>f</code>	function to integrate
<code>a</code>	the a-bound of integration
<code>b</code>	the b-bound of integration
<code>m</code>	the number of subintervals to calculate

**Details**

The `simp38` function uses Simpson's 3/8 rule to calculate the integral of the function `f` over the interval from `a` to `b`. The parameter `m` sets the number of intervals to use when evaluating. Additional options are passed to the function `f` when evaluating.

**Value**

the value of the integral

**See Also**

Other integration: [adaptint\(\)](#), [gaussint\(\)](#), [giniquintile\(\)](#), [mcint\(\)](#), [midpt\(\)](#), [revolution-solid](#), [romberg\(\)](#), [simp\(\)](#), [trap\(\)](#)

Other newton-cotes: [adaptint\(\)](#), [giniquintile\(\)](#), [midpt\(\)](#), [romberg\(\)](#), [simp\(\)](#), [trap\(\)](#)

**Examples**

```
f <- function(x) { sin(x)^2 + log(x) }
simp38(f, 1, 10, m = 10)
simp38(f, 1, 10, m = 100)
simp38(f, 1, 10, m = 1000)
```

---

`summation`*Two summing algorithms*

---

**Description**

Find the sum of a vector

**Usage**`naivesum(x)``kahansum(x)``pwisum(x)`**Arguments**

`x` a vector of numbers to be summed

**Details**

`naivesum` calculates the sum of a vector by keeping a counter and repeatedly adding the next value to the interim sum. `kahansum` uses Kahan's algorithm to capture the low-order precision loss and ensure that the loss is reintegrated into the final sum. `pwisum` is a recursive implementation of the piecewise summation algorithm that divides the vector in two and adds the individual vector sums for a result.

**Value**

the sum

**Examples**

```
k <- 1:10^6
n <- sample(k, 1)
bound <- sample(k, 2)
bound.upper <- max(bound) - 10^6 / 2
bound.lower <- min(bound) - 10^6 / 2
x <- runif(n, bound.lower, bound.upper)
naivesum(x)
kahansum(x)
pwisum(x)
```

---

trap

*Trapezoid method*

---

### Description

Use the trapezoid method to integrate a function

### Usage

```
trap(f, a, b, m = 100)
```

### Arguments

f	function to integrate
a	the a-bound of integration
b	the b-bound of integration
m	the number of subintervals to calculate

### Details

The trap function uses the trapezoid method to calculate the integral of the function f over the interval from a to b. The parameter m sets the number of intervals to use when evaluating the trapezoids. Additional options are passed to the function f when evaluating.

### Value

the value of the integral

### See Also

Other integration: [adaptint\(\)](#), [gaussint\(\)](#), [giniquintile\(\)](#), [mcint\(\)](#), [midpt\(\)](#), [revolution-solid](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#)

Other newton-cotes: [adaptint\(\)](#), [giniquintile\(\)](#), [midpt\(\)](#), [romberg\(\)](#), [simp38\(\)](#), [simp\(\)](#)

### Examples

```
f <- function(x) { sin(x)^2 + cos(x)^2 }
trap(f, -pi, pi, m = 10)
trap(f, -pi, pi, m = 100)
trap(f, -pi, pi, m = 1000)
```

---

tridiagmatrix	<i>Solve a tridiagonal matrix</i>
---------------	-----------------------------------

---

**Description**

use the tridiagonal matrix algorithm to solve a tridiagonal matrix

**Usage**

```
tridiagmatrix(L, D, U, b)
```

**Arguments**

L	vector of entries below the main diagonal
D	vector of entries on the main diagonal
U	vector of entries above the main diagonal
b	vector of the right-hand side of the linear system

**Details**

tridiagmatrix uses the tridiagonal matrix algorithm to solve a tridiagonal matrix.

**Value**

the solution vector

**See Also**

Other linear: [choleskymatrix\(\)](#), [detmatrix\(\)](#), [gdls\(\)](#), [invmatrix\(\)](#), [iterativematrix](#), [lumatrix\(\)](#), [refmatrix\(\)](#), [rowops](#), [vecnorm\(\)](#)

---

vecnorm	<i>Norm of a vector</i>
---------	-------------------------

---

**Description**

Find the norm of a vector

**Usage**

```
vecnorm(b)
```

**Arguments**

b	a vector
---	----------

**Details**

Find the norm of a vector

**Value**

the norm

**See Also**

Other linear: [choleskymatrix\(\)](#), [detmatrix\(\)](#), [gdls\(\)](#), [invmatrix\(\)](#), [iterativematrix](#), [lumatrix\(\)](#), [refmatrix\(\)](#), [rowops](#), [tridiagmatrix\(\)](#)

**Examples**

```
x <- c(1, 2, 3)
vecnorm(x)
```

---

wave

*Wave Equation using*

---

**Description**

solve heat equation via forward-time central-space method

**Usage**

```
wave(u, alpha, xdelta, tdelta, n)
```

**Arguments**

u	the initial values of u
alpha	the thermal diffusivity coefficient
xdelta	the change in x at each step in u
tdelta	the time step
n	the number of steps to take

**Details**

The heat solves the heat equation using the forward-time central-space method in one-dimension.

**Value**

a matrix of u values at each time step

**Examples**

```
speed <- 2
x0 <- 0
xdelta <- .05
x <- seq(x0, 1, xdelta)
m <- length(x)
u <- sin(x * pi * 2)
u[11:21] <- 0
tdelta <- .02
n <- 40
z <- wave(u, speed, xdelta, tdelta, n)
```

---

wilkinson

*Wilkinson's Polynomial*

---

**Description**

Wilkinson's polynomial

**Usage**

```
wilkinson(x, w = 20)
```

**Arguments**

x	the x-value
w	the number of terms in the polynomial

**Details**

Wilkinson's polynomial is a terrible joke played on numerical analysis. By tradition, the function is  $f(x) = (x - 1)(x - 2)\dots(x - 20)$ , giving a function with real roots at each integer from 1 to 20. This function is generalized and allows for  $n$  and the function value is  $f(x) = (x - 1)(x - 2)\dots(x - n)$ . The default of  $n$  is 20.

**Value**

the value of the function at  $x$

**Examples**

```
wilkinson(0)
```

# Index

- \* **adaptive**
    - adaptint, 3
  - \* **algebra**
    - bilinear, 5
    - cubicspline, 8
    - division, 10
    - fibonacci, 11
    - horner, 20
    - isPrime, 22
    - linterp, 25
    - nthroot, 31
    - polyinterp, 32
    - pwiselinterp, 33
    - quadratic, 34
  - \* **differentiation**
    - findiff, 12
  - \* **integration**
    - adaptint, 3
    - gaussint, 13
    - giniquintile, 15
    - mcint, 27
    - midpt, 28
    - revolution-solid, 36
    - romberg, 37
    - simp, 41
    - simp38, 42
    - trap, 44
  - \* **interpolation**
    - resizeImage, 36
  - \* **interp**
    - bezier, 4
    - bilinear, 5
    - cubicspline, 8
    - linterp, 25
    - nn, 30
    - polyinterp, 32
    - pwiselinterp, 33
  - \* **intro**
    - summation, 43
  - \* **linear**
    - choleskymatrix, 8
    - detmatrix, 9
    - gdls, 14
    - invmatrix, 21
    - iterativematrix, 23
    - lumatrix, 26
    - refmatrix, 35
    - rowops, 38
    - tridiagmatrix, 45
    - vecnorm, 45
  - \* **newton-cotes**
    - adaptint, 3
    - giniquintile, 15
    - midpt, 28
    - romberg, 37
    - simp, 41
    - simp38, 42
    - trap, 44
  - \* **optimz**
    - bisection, 6
    - goldsect, 16
    - gradient, 17
    - hillclimbing, 19
    - newton, 29
    - sa, 39
    - secant, 40
  - \* **polynomials**
    - wilkinson, 47
- adamsbashforth (ivp), 24
- adaptint, 3, 13, 15, 27, 28, 37, 38, 41, 42, 44
- betterpoly (horner), 20
- bezier, 4, 6, 9, 26, 30, 32, 33
- bilinear, 5, 5, 9–11, 21, 22, 26, 30–34
- bisection, 6, 16, 17, 19, 29, 39, 40
- bvp, 7
- bvpexample (bvp), 7
- bvpexample10 (bvp), 7



- cbezier (bezier), 4
- cgmmatrix (iterativematrix), 23
- choleskymatrix, 8, 10, 14, 22, 23, 27, 35, 39, 45, 46
- cmna (cmna-package), 3
- cmna-package, 3
- cubicspline, 5, 6, 8, 10, 11, 21, 22, 26, 30–34
- detmatrix, 8, 9, 14, 22, 23, 27, 35, 39, 45, 46
- discmethod (revolution-solid), 36
- division, 6, 9, 10, 11, 21, 22, 26, 31–34
- euler (ivp), 24
- eulersys (ivpsys), 25
- fibonacci, 6, 9, 10, 11, 21, 22, 26, 31–34
- findiff, 12
- findiff2 (findiff), 12
- gauss.hermite (gaussint), 13
- gauss.laguerre (gaussint), 13
- gauss.legendre (gaussint), 13
- gaussint, 4, 13, 15, 27, 28, 37, 38, 41, 42, 44
- gaussseidel (iterativematrix), 23
- gd (gradient), 17
- gdls, 8, 10, 14, 22, 23, 27, 35, 39, 45, 46
- giniquintile, 4, 13, 15, 27, 28, 37, 38, 41, 42, 44
- goldsect, 6, 16, 17, 19, 29, 39, 40
- goldsectmax (goldsect), 16
- goldsectmin (goldsect), 16
- gradasc (gradient), 17
- graddsc (gradient), 17
- gradient, 6, 16, 17, 19, 29, 39, 40
- heat, 18
- hillclimbing, 6, 16, 17, 19, 29, 39, 40
- himmelblau, 20
- horner, 6, 9–11, 20, 22, 26, 31–34
- invmatrix, 8, 10, 14, 21, 23, 27, 35, 39, 45, 46
- isPrime, 6, 9–11, 21, 22, 26, 31–34
- iterativematrix, 8, 10, 14, 22, 23, 27, 35, 39, 45, 46
- ivp, 24
- ivpsys, 25
- jacobi (iterativematrix), 23
- kahansum (summation), 43
- linterp, 5, 6, 9–11, 21, 22, 25, 30–34
- longdiv (division), 10
- lumatrix, 8, 10, 14, 22, 23, 26, 35, 39, 45, 46
- mcint, 4, 13, 15, 27, 28, 37, 38, 41, 42, 44
- mcint2 (mcint), 27
- midpt, 4, 13, 15, 27, 28, 37, 38, 41, 42, 44
- midptivp (ivp), 24
- naivediv (division), 10
- naivepoly (horner), 20
- naivesum (summation), 43
- newton, 6, 16, 17, 19, 29, 39, 40
- nn, 5, 6, 9, 26, 30, 32, 33
- nthroot, 6, 9–11, 21, 22, 26, 31, 32–34
- polyinterp, 5, 6, 9–11, 21, 22, 26, 30, 31, 32, 33, 34
- pwiselinterp, 5, 6, 9–11, 21, 22, 26, 30–32, 33, 34
- pwisesum (summation), 43
- qbezier (bezier), 4
- quadratic, 6, 9–11, 21, 22, 26, 31–33, 34
- quadratic2 (quadratic), 34
- rdiff (findiff), 12
- refmatrix, 8, 10, 14, 22, 23, 27, 35, 39, 45, 46
- replacerow (rowops), 38
- resizeImage, 36
- resizeImageBL (resizeImage), 36
- resizeImageNN (resizeImage), 36
- revolution-solid, 36
- rhorners (horner), 20
- romberg, 4, 13, 15, 27, 28, 37, 37, 41, 42, 44
- rowops, 8, 10, 14, 22, 23, 27, 35, 38, 45, 46
- rrefmatrix (refmatrix), 35
- rungekutta4 (ivp), 24
- sa, 6, 16, 17, 19, 29, 39, 40
- scalerow (rowops), 38
- secant, 6, 16, 17, 19, 29, 39, 40
- shellmethod (revolution-solid), 36
- simp, 4, 13, 15, 27, 28, 37, 38, 41, 42, 44
- simp38, 4, 13, 15, 27, 28, 37, 38, 41, 42, 44
- solvematrix (refmatrix), 35
- summation, 43
- swaprows (rowops), 38
- syndiff (findiff), 12

trap, [4](#), [13](#), [15](#), [27](#), [28](#), [37](#), [38](#), [41](#), [42](#), [44](#)  
tridiagmatrix, [8](#), [10](#), [14](#), [22](#), [23](#), [27](#), [35](#), [39](#),  
[45](#), [46](#)  
tspas (sa), [39](#)  
  
vecnorm, [8](#), [10](#), [14](#), [22](#), [23](#), [27](#), [35](#), [39](#), [45](#), [45](#)  
  
wave, [46](#)  
wilkinson, [47](#)